CSE 160
Lecture 2

Programming with Threads
Parallel Sorting

Scott B. Baden

Announcements

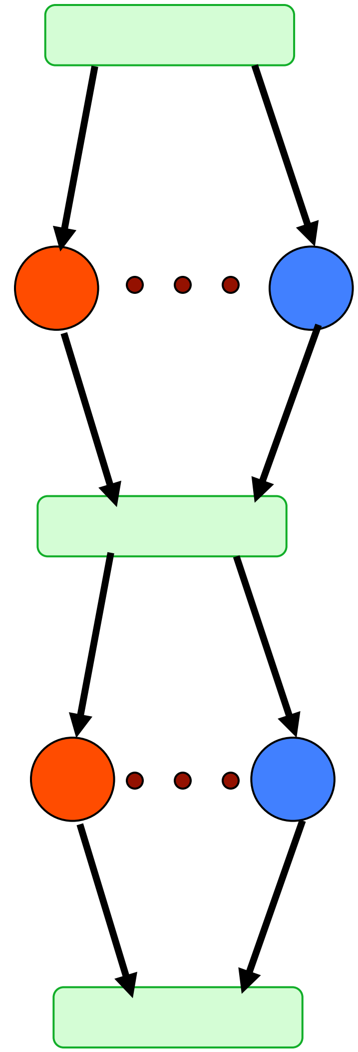
- Makeup on 10/7
- Quiz #1 on Weds 10/9
- SVN

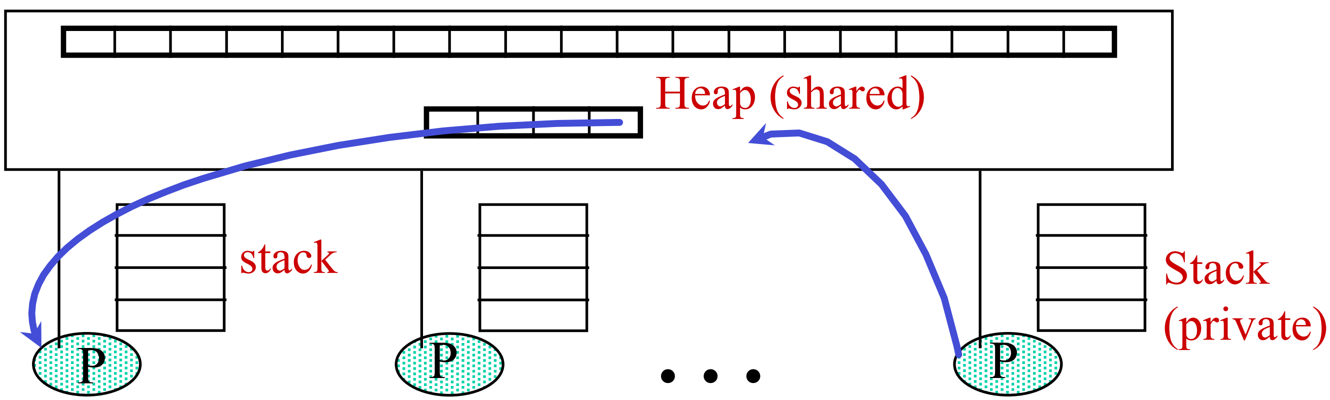
Today's lecture

- Two applications with multithreading
- Synchronization
- Parallel Sorting

Recall the Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
- Threads may or may not execute on different processors





C++11 Threads

- Via <thread>, C++ supports a threading interface similar to pthreads, though a bit more user friendly
- Async is a higher level interface suitable for certain kinds of applications
- New memory model
- Atomic template

Hello world with <Threads>

```
#include <thread>
void Hello(int TID) {
    cout << "Hello from thread " <<   TID << endl;
}

int main(int argc, char *argv[ ]){
  thread *thrds = new thread[NT];

// Spawn threads
for(int t=0;t<NT;t++){
     thrds[t] = thread(Hello, t );
}

// Join threads
for(int t=0;t<NT;t++)
      thrds[t].join();
}
```

```
$ ./hello_th 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
$ ./hello_th 3
Hello from thread 1
Hello from thread 0
Hello from thread 2
$ ./hello_th 4
Running with 4 threads
Hello from thread 0
Hello from thread 3
Hello from thread Hello from
thread 21
```

$PUB/Examples//Threads/Hello-Th

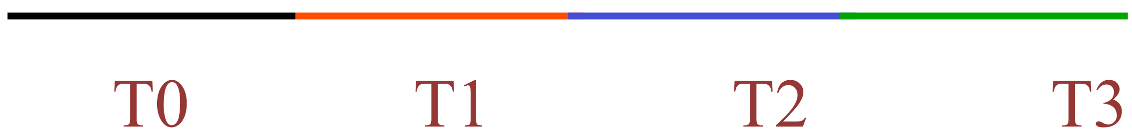
PUB = /share/class/public/cse160-fa13

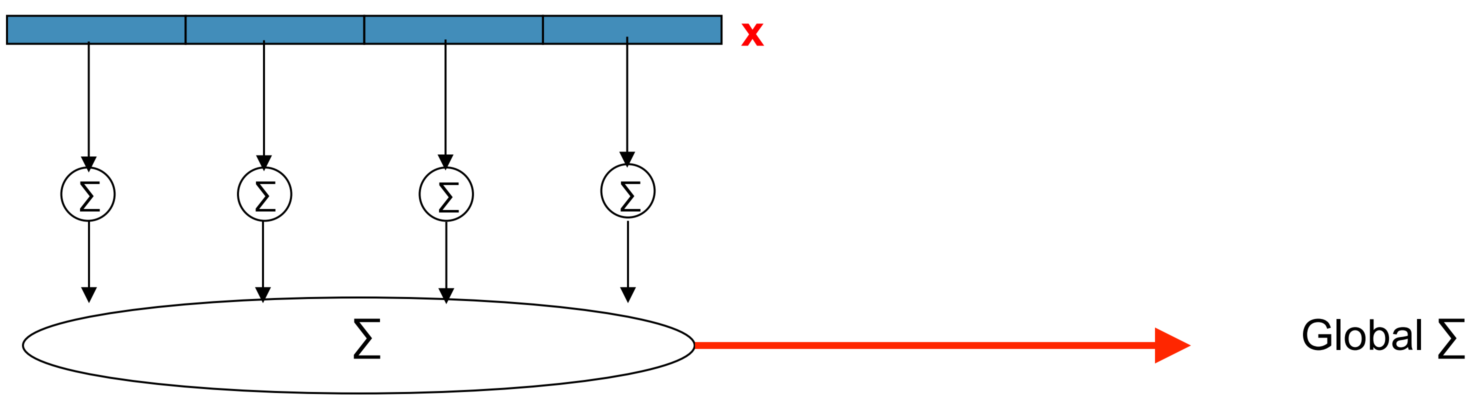
Steps in writing multithreaded code

- We write a *thread function* that gets called each time we spawn a new thread
- *Spawn* threads by constructing objects of class Thread (in the C++ library)
- Each thread runs on a separate processing core (If more threads than cores, the threads share cores)
- *Join* threads so we know when they are done

A first application

- Sum a list of integers
```
for i = 0:N-1
      sum = sum + x[i];
```
- Partition x[] into intervals, assign each to a unique thread
- Each thread sweeps over a reduced problem





First version of summing code

```
void sum(int TID, int N, int NT){
   int64_t i0 = TID*(N/NT),  i1   = i0 + (N/NT);
   int64_t local_sum=0;
   for (int i=i0;  i<i1;  i++)
      local_sum += x[i];
   global_sum += local_sum
}
```

```
 int* x;
Main():
 int64_t global_sum;
 for(int t=0; t<NT; t++){
    thrds[t] = thread(sum,t,N,NT);
```

Steps in writing multithreaded code (II)

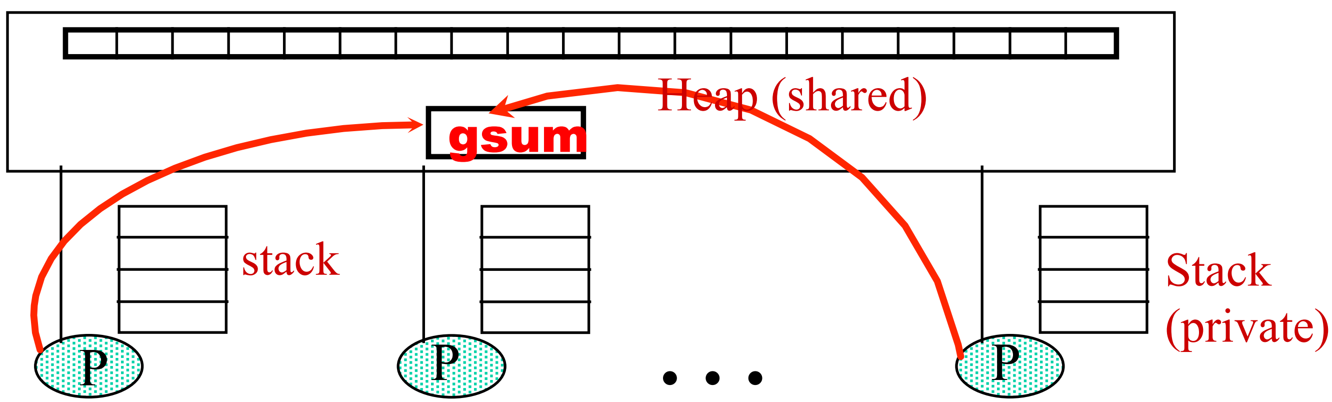
- We write a *thread function* that gets called each time we spawn a new thread
- *Spawn* threads by constructing objects of class Thread (in the C++ library)
- Each thread runs on a separate processing core
(If more threads than cores, the threads share cores)
- *Join* threads so we know when they are done
- Threads share memory

Today's lecture

- Two applications with multithreading
- Synchronization
- Parallel Sorting

Results

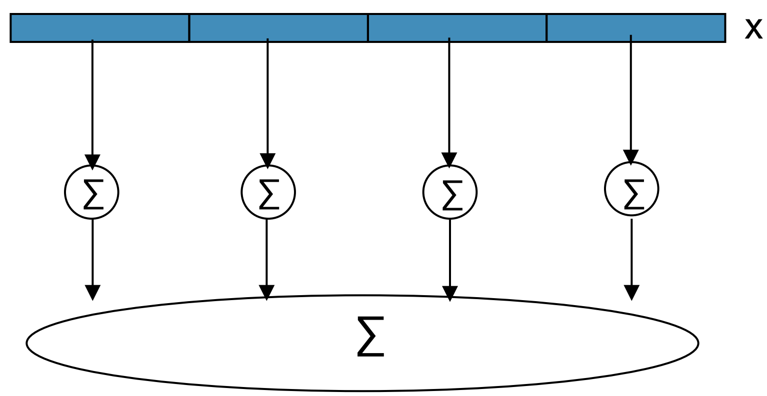
- The program usually runs correctly
- But sometimes it produces incorrect results:
  Result verified to be INCORRECT, should be 549756338176
- What happened?
- There is a conflict when updating global_sum: a *data race*



Data race

- A date race arises when there is at least one writer on shared data
- There are multiple writers of global_sum

```
int64_t global_sum;
void sum(int TID, int N, int NT){
    int64_t i0 = TID*(N/NT),  i1   = i0 + (N/NT);
    int64_t localSum=0;
    for (int i=i0;  i<i1;  i++)
        localSum += x[i];
    global_sum += local_sum
}
```



Avoiding the data race

- Perform the global summation in main()
- After a thread joins, add its contribution to the global sum, one thread at a time
- We need to wrap ref() around ref arguments, int64_t &, compiler needs the hint

```
int64_t global_sum, local_sum;
 …
int *locSims = new int[NT];
for(int t=0; t<NT; t++)
   thrds[t] = thread(sum,t,N,NT,ref(locSums[t]);
for(int t=0; t<NT; t++){
   thrds[t].join();
   global_sum += local_sum;
}
```

Steps in writing multithreaded code (III)

- We write a *thread function* that gets called each time we spawn a new thread
- *Spawn* threads by constructing objects of class Thread (in the C++ library)
- Each thread runs on a separate processing core
(If more threads than cores, the threads share cores)
- *Join* threads so we know when they are done
- Threads share memory
- Avoid data races to ensure correctness

Race conditions

- Consider the following thread function, where x is initially 0

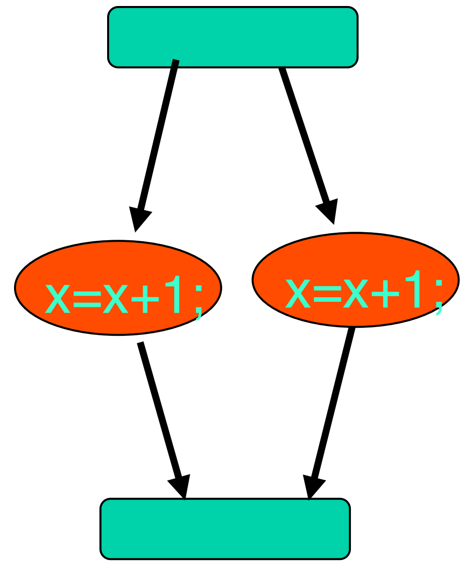
```
void threadFn(int TID) {
    x++;
}
```

- Let run on 2 threads
- What is the value of x after both threads have joined?
- A *race condition* arises because the timing of accesses to shared data can affect the outcome
- We say we have a *non-deterministic* computation
- Normally, if we repeat a computation using the same inputs we expect to obtain the same results
- This is true because we have a *side effect* (global variables, I/O and random number generators)

Under the hood of a race condition

- Assume x is initially 0

  x=x+1;



- Generated assembly code
  - r1 ← (x)
  - r1 ← r1 + #1
  - r1 → (x)

- Possible interleaving with two threads

P1	P2	
r1 ← x		*r1(P1) gets 0*
	r1 ← x	*r2(P2) also gets 0*
r1 ← r1+ #1		*r1(P1) set to 1*
	r1 ← r1+#1	*r1(P1) set to 1*
x ← r1		*P1 writes its R1*
	x ← r1	*P2 writes its R1*

Avoiding race conditions

- We need to take steps to avoid race conditions through appropriate program synchronization
  - Migrate shared updates into main
  - Critical sections
  - Barriers
  - Atomics

Critical Sections

- In some cases it is costly (or inconvenient) to join and re-spawn threads to synchronize
- Instead, we synchronize inside the thread function
- We must allow only 1 thread at a time to write to the shared memory location(s)
- The code performing the operation is called a *critical section*
- We use *mutual exclusion* to implement a critical section
- A critical section is non-parallelizing computation.. sensible guidelines?

```
Begin Critical Section
    x++;
End Critical Section
```

Using mutexes in C++

- The <mutex> library provides a mutex class
- A mutex (AKA a "lock") may be CLEAR or SET
  - Lock() waits if the lock is set, else sets the lock
  - Unlock() clears the lock if set
- Mutexes are global variables. Why?

```
void sum(int TID, int N, int NT){
   …
   for (int i=i0;  i<i1;  i++)
       localSum += x[i];
// Critical section
   mutex_sum.lock();
   global_sum += localSum;
   mutex_sum.unlock();
}
```

```
int* x;
  mutex mutex_sum;
  int64_t global_sum;
Main():
// Spawn threads
```

Results

- ./sum 1 1000000000
  1.30 seconds
- ./sum 2 10^9
  0.79 seconds [speedup = 1.64]
- ./sum 4 10^9
  0.69 seconds [incremental speedup = 1.14]
- ./sum 8 10^9
  0.68 seconds [incremental speedup = 1.01]

Using a more expensive kernel

- `for (int i=i0; i<i1; i++)`
  `sum += sin(x[i]);`
- ./sumSine 1 10^8
  6.50 seconds
- ./sumSine 2 10^8
  3.27 seconds [speedup = 1.99]
- ./sumSine 4 10^8
  1.63 seconds [incremental speedup = 2.0]
- ./sumSine 8 10^8
  0.82 seconds [incremental speedup = 1.99]

How do we explain the results?

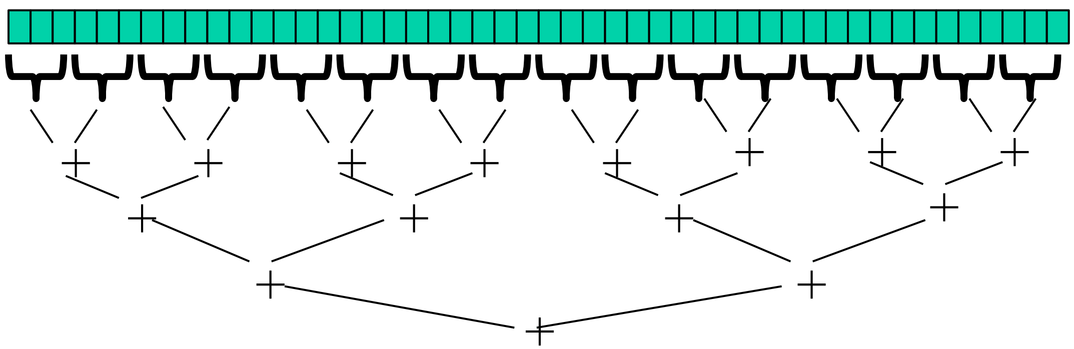
- Expensive kernel gets perfect speedup on 4 cores
- Inexpensive kernel gets a speedup of 1.9

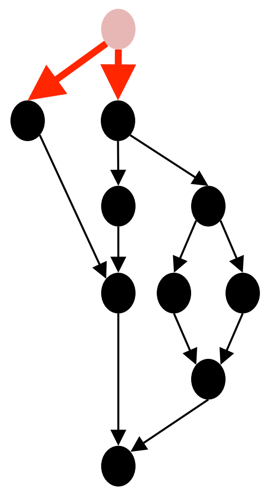
2nd application: testing for primality

- Given a list of numbers, which are prime?
  primes <# threads> 2 17 31 3415501328329
- Code in $PUB/Examples/Threads/Primes
- 3 Versions: Threads, Async (later), Pthreads

Other kinds of threading structures

- We may create elaborate threading structures, for example, divide and conquer





Steve Wolfman, based on work by Dan Grossman

Today's lecture

- Two applications with multithreading
- Synchronization
- Parallel Sorting

Parallel Sorting

- Sorting is fundamental algorithm in data processing
  - Given an unordered set of keys $x_0, x_1, \ldots, x_{N-1}$
  - Return the keys in sorted order
- The keys may be character strings, floating point numbers, integers, or any object for which the relations >, <, and == hold
- We'll assume integers here
- Will talk about other algorithms later on

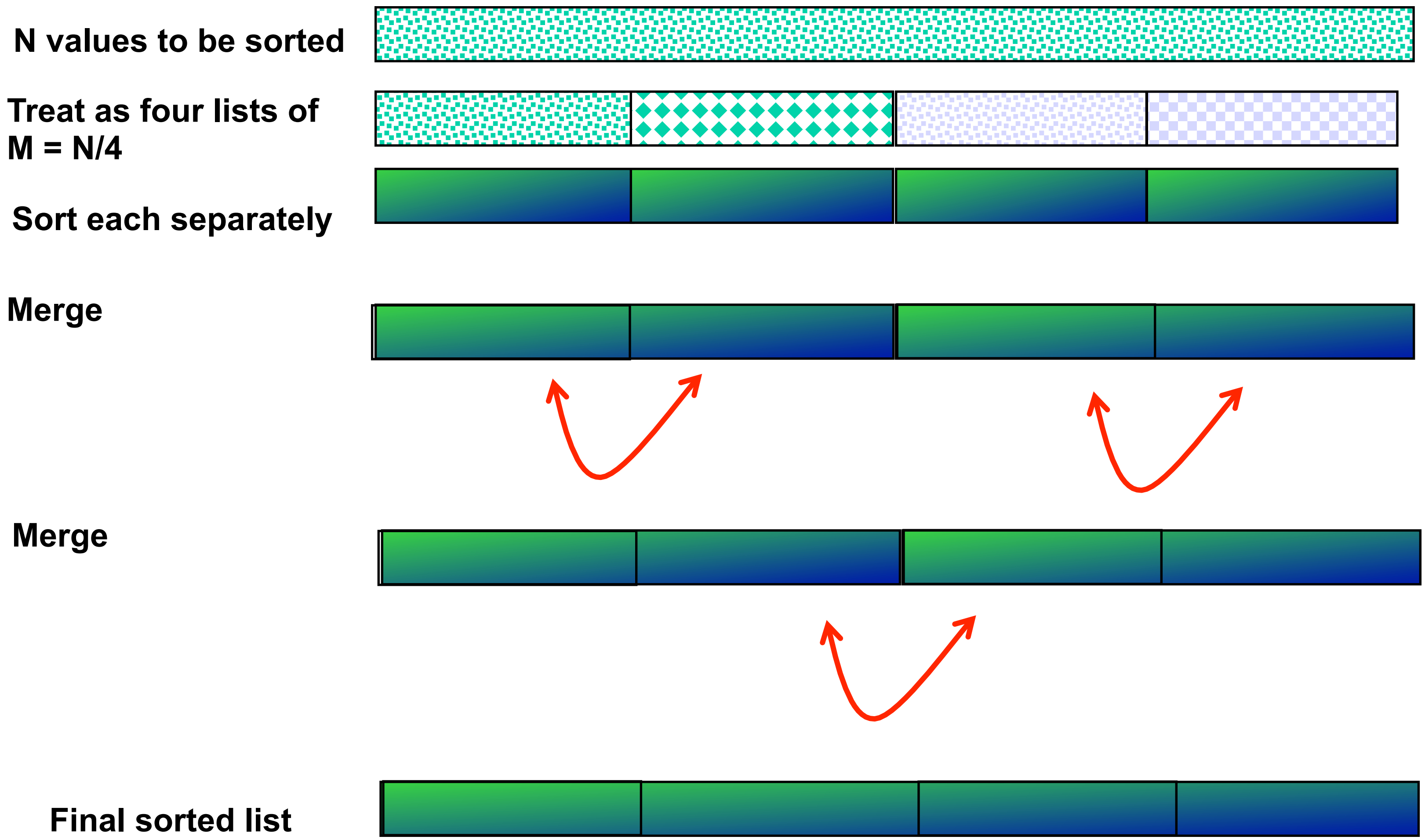
Parallel sorting algorithms

- We'll consider in-memory sorting of integer keys
  - Merge Sort
  - Bucket sort
  - Sample sort
  - Bitonic sort
- In practice, we sort on external media, i.e. disk
  - See: http://sortbenchmark.org
  - **TritonSort (UCSD):** 0.725×10^{12} bytes/minute

Merge Sort algorithm

- A divide and conquer algorithm
- When we reach a certain size, we stop the recursion: each thread locally sorts its data using a fast serial algorithm like quicksort
- Threads merge their data in odd-even pairs
- Each thread applies a local merge sort to extract the smallest (largest) N/P values, discards the rest
- What is the running time?

Merge sort in action



Serial Merge

-1 3 7 9 11 2 4 8 12 14

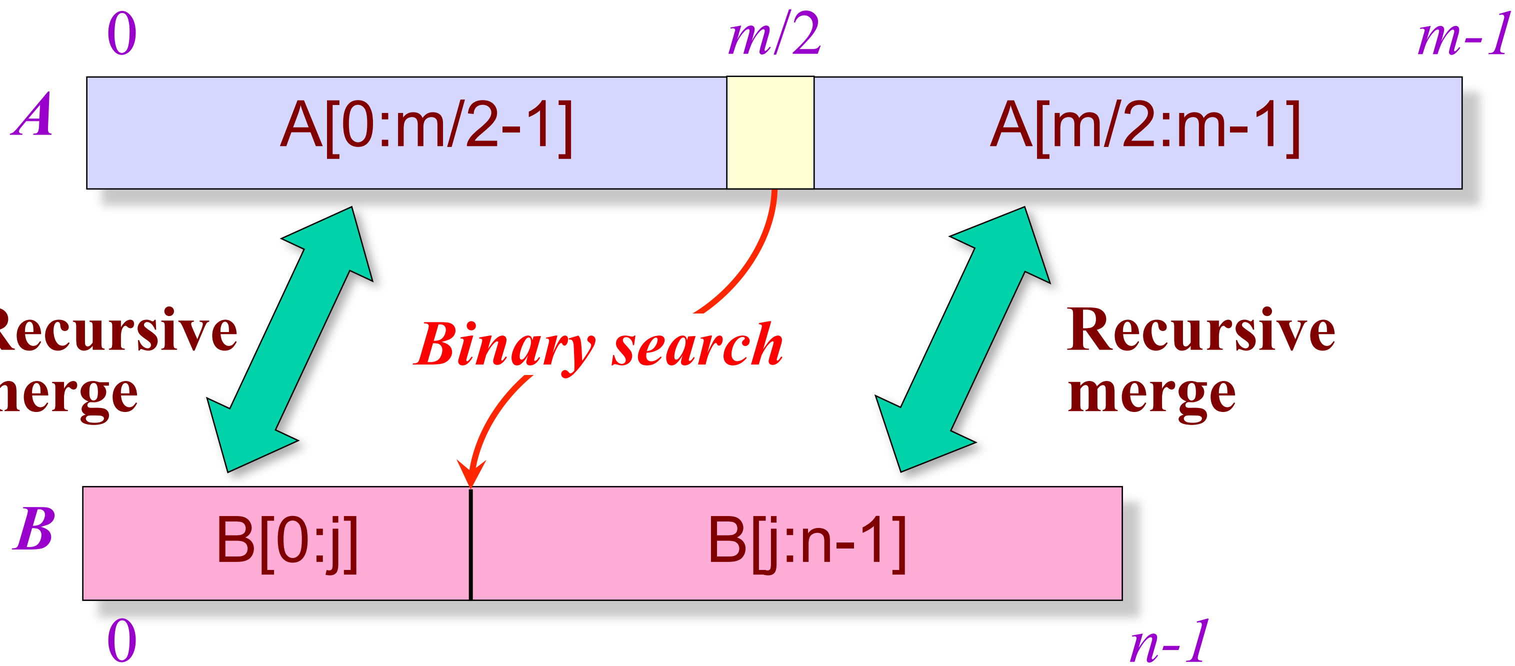
Thread 0 Thread 1

- Merge Step
- Left most thread does the merging
  -1 3 7 9 11 2 4 8 12 14
- Sorts the merged list
  -1 2 3 4 7 8 9 11 2 14

- Parallelism diminishes as we move up the recursion tree
- There is only O(log n) parallelism, but if we stop the recursion before reaching the bottom of the tree, it's much smaller

Parallel Merge

- If there are $N = m+n$ elements, then the larger of the recursive merges processes $\frac{3}{4}N$ elements
- Assume $m \geq n$ (switch arrays if necessary)



Charles Leiserson

Assignment #1

- Implement parallel merge sort
- Implement parallel merge and determine how much it helps
- Observe speedups
- Develop on Ieng6, benchmarking on Bang
- Use SVN for you code development
  - Starter code available via SVN
  - Required to use SVN repository on Bang
  - Do not use github or other repositories
  - Any sharing of code is a breach of Academic Integrity
  - SVN Discussion in section on Wednesday
  - Be sure to respond to posting about registering your team
- A4 will be posted by Wednesday evening